Reprinted from the MATHEMATICS MAGAZINE
Vol. 58, No. 1, January 1985

Thirty Days Hath February

BERNARD E. RODDEN
Defence and Civil Institute of Environmental Medicine
Downsview, Ontario, Canada M3M 3B9

Algorithms pertaining to the Gregorian calendar system provide a pool of practical classroom problems for lectures in computation. Everyone has a common sense appreciation of the calendar, and associated algorithms have sufficient scope to be entertaining without being intellectually demanding. Unfortunately, it is sometimes difficult to assess the quality of a given algorithm, as criteria of "goodness" are typically understated, and left to the reader's imagination (on the mistaken assumption that such criteria are self-evident).

A typical date algorithm is the production of a yearly calendar for any given year. Can reasonable people agree on what are *good* properties for a calendar algorithm? I suggest that the most obvious good property is the restriction to integer arithmetic. As the calendar is defined by integers, we should not have to leave the integer system to solve date problems. Computers work faster with integers; and integer algorithms are easily implemented in firmware. Not so obvious is the property that dates should be *generated* rather than be dependent on tabulated information. *Tables* in this sense include such computer language constructs as *arrays*, *structures*, *records*, *cases*, and *types*. Tables are usually relatively awkward and slow to manipulate, error prone, and difficult to transport from one system or language to another. More to the point, tables only assist in a mapping from one set of integers to another—an operation that can often be done more elegantly and efficiently by an algorithm. Many date algorithms presently in use apply only to a portion of the full period of the calendar. Since the full period is only 28,000 years, surely it is not too demanding to insist that date algorithms be effective over the entire cycle. The final good property of an algorithm is the ease with which it can be coded efficiently using modern computing languages.

In this Note, I shall describe a calendar generator based on ideas contained in an Algol procedure by R. A. Stone [1]. Stone used three particular integer parameters to map the days of a standardized year onto the days of the month. I use his three-parameter formulation, the standardized year, and his emphasis on tableless algorithms. In addition, I extend his three parameters to an infinite class, develop an inverse to his function, and show that the method can be used for any date problem.

The Gregorian calendar as it is today (with the exception of the 4000-year rule) was introduced in a papal bull dated March 1, 1582 [2]. If we extrapolate the current calendar backwards, this date would be March 11, because the *new style* calendar was to become effective on October 15 1582, replacing the *old style* date of October 5. The weekday sequence was unaltered. The only significant change from a mathematical point of view was the introduction of a rule excluding century years from the set of leap years, unless they were divisible by 400. The effect of these exclusions was to reduce the accumulation of error with respect to the solar year from one day every 128 years (approximately) to one day every 3323 years. This modification was fortuitous because a 400-year interval (146,097 days) is divisible by 7, and hence the period of calendar dates coincided with the approximation to the astronomic period. The most recent, and likely final, proposed correction fits the period of the earth's revolution to a 4000-year interval consisting of 1,460,969 days, which is not divisible by 7, and hence the calendar period is 28,000 years. The 4000-year rule reduces the error to one day every 20,000 years.

The (year, month, day) mixed radix system of the calendar is more complex than some other mixed systems, such as (pounds, shillings, pence), because the radix for the number of days is not fixed, but is a function of both month and year. An additional complication is the weekday cycle. There are 7 weekdays and two possible types of year, leap and non-leap, providing 14 distinct, yearly calendars.

85-3806/

A calendar can be generated for any year y by the following sequence of operations. Let (y, m, d) describe day d of month m of year y. By definition, $(y, 1, 1)$ is a valid date. The weekday corresponding to $(y, 1, 1)$ can be deduced from application of the weekday theorem, which we develop later. Let $(y, m, d)^*$ be the day subsequent to (y, m, d). If (y, m, d) is a valid date, then $(y, m, d)^*$ is the first valid member of the ordered list $\{(y, m, d+1), (y, m+1, 1), (y+1, 1, 1)\}$. As the third member is always valid, there is always a valid subsequent date. Hence we only require a criterion for validating a given date. The criterion may be expressed as

$$(y, m, d) \text{ is a valid date if and only if } 0 < d \leq L(m, q),$$

where m is a valid month number, year y is of type q, and $L(m, q)$ is the number of days in month m of year type q.

If we can find a functional form of L, along with the weekday function, we will have a tableless calendar generator.

The integer y represents a leap year if and only if y is a multiple of 4 but not of 100, or y is a multiple of 400 but not of 4000. Although this definition of a leap year seems a natural verbal description, it can lead to inefficient algorithms, since it does not take full advantage of the probabilistic ordering of the various conditions. Simplistic implementation results in the logical function

$$q(y) = [D_4(y) \cap \overline{D}_{100}(y)] \cup [D_{400}(y) \cap \overline{D}_{4000}(y)] \tag{1}$$

where $D_n(y)$ is *true* if y is a multiple of n, and *false* otherwise, and $\overline{D}_n(y)$ denotes the negation of $D_n(y)$. Hence q is *true* if y is a leap year, and *false* otherwise, as shown in the Venn diagram of FIGURE 1.

In this algorithm, the D calculations are computationally relatively expensive, and we shall show that despite partial optimization, the simplistic form (1) increases the expected number of D calculations about 50% with respect to a most efficient algorithm. We define a *most efficient* algorithm as one that requires the smallest expected number of D calculations for a year selected randomly from a full cycle of the calendar. To design such an algorithm, we take advantage of the sequence of logical implications,

$$D_{4000} \subset D_{400} \subset D_{100} \subset D_4$$

by so ordering the sequence of logical operations of the algorithm for $q(y)$ that those events that most probably determine q are calculated first. For example, the *truth* of D_{4000} establishes immediately the type of year, but D_{4000} is a rare event; the *falsity* of D_4 also establishes the type of year, and is the most probable event that does so. The efficient order of calculation is thus:

$$\overline{D}_4, \overline{D}_{100} | D_4, \overline{D}_{400} | D_{100}, \overline{D}_{4000} | D_{400},$$

where $\overline{D}_m | D_n$ means $\overline{D}_m$ given D_n.

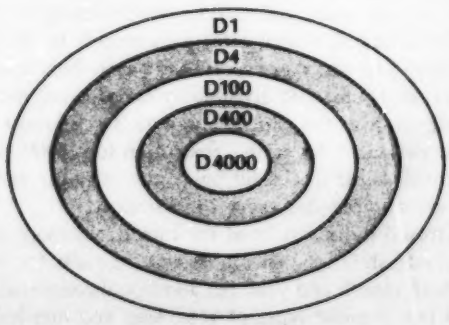

FIGURE 1. Venn diagram of years.

In a corresponding algorithm

$$q(y) = D_4 \cap [\overline{D}_{100} \cup (D_{400} \cap \overline{D}_{4000})] \tag{2}$$

calculation is from left to right, and ceases when the truth or falsity of the expression has been established. Hence for the determination of q, using (2), the expected number of D calculations is

$$p(\overline{D}_4)\cdot 1 + p(D_4)\cdot[p(\overline{D}_{100}|D_4)\cdot 2 + p(D_{100}|D_4)\cdot[p(\overline{D}_{400}|D_{100})\cdot 3 + p(D_{400}|D_{100})\cdot 4]]$$

$$= \frac{3}{4}\cdot 1 + \frac{1}{4}\cdot\left(\frac{24}{25}\cdot 2 + \frac{1}{25}\cdot\left(\frac{3}{4}\cdot 3 + \frac{1}{4}\cdot 4\right)\right)$$

$$= 1.2625,$$

where $p(X|Y)$ means the conditional probability of event X given event Y.

The corresponding expansion for the simplistic form (1) gives an expected number of D calculations

$$[p(D_4)\cdot p(\overline{D}_{100}|D_4) + p(\overline{D}_4)\cdot p(\overline{D}_{400}|\overline{D}_4)]\cdot 2$$
$$+ p(D_4)\cdot p(D_{100}|D_4)\cdot[p(\overline{D}_{400}|D_{100})\cdot 3 + p(D_{400}|D_{100})\cdot 4]$$
$$= \left(\frac{1}{4}\cdot\frac{24}{25} + \frac{3}{4}\cdot 1\right)\cdot 2 + \frac{1}{4}\cdot\frac{1}{25}\cdot\left(\frac{3}{4}\cdot 3 + \frac{1}{4}\cdot 4\right)$$
$$= 2.0125.$$

Many computer languages do not interpret the left-to-right logical operations in the efficient manner described. For such languages, the coder must break down the algorithm in the obvious way, and branch out explicitly. Some languages, such as *C* and Modula, allow direct expression.

To calculate the day of the week on which an arbitrary date falls, we first compute the number of leap years between the given date and a standard date. To simplify the calculation of the number of leap year days between two arbitrary dates, we define **the number of leap year days** $n(y)$ to be the *algebraic* difference between year 0 and date $(y,1,1)$, as given by the following formulas.

$$\text{If } y > 0, \quad n(y) = \left\lfloor \frac{y-1}{4} \right\rfloor - \left\lfloor \frac{y-1}{100} \right\rfloor + \left\lfloor \frac{y-1}{400} \right\rfloor - \left\lfloor \frac{y-1}{4000} \right\rfloor \tag{3}$$

$$\text{If } y < 0, \quad n(y) = \left\lceil \frac{y}{4} \right\rceil - \left\lceil \frac{y}{100} \right\rceil + \left\lceil \frac{y}{400} \right\rceil - \left\lceil \frac{y}{4000} \right\rceil. \tag{4}$$

where $\lfloor x \rfloor$ is the floor function, that is, $\lfloor x \rfloor$ is the greatest integer not greater than the number x, and where $\lceil x \rceil$ is the ceiling function, that is, $\lceil x \rceil$ is the least integer not less than the number x. There is no computational complexity in using the floor function for positive quotients and the ceiling function for negative quotients, as this is the definition of integer division for most computer languages. As an example, the number of leap year days between the dates $(-4,1,1)$ and $(4,1,1)$ is $n(4) - n(-4) = 0 - (-1) = 1$.

We are now able to calculate the weekday corresponding to the first day of any year.

THE WEEKDAY THEOREM. *Year y begins on weekday w if*

$$w = (n + 365y)_7, \quad \text{or } w = (n + y)_7, \quad \text{or } w = (n + (y)_7)_7, \tag{5}$$

where $(x)_7$ is the member of the set $\{0,1,\ldots,6\}$ congruent modulo 7 to x, and where $n = n(y)$ is the number of leap years defined by (3) and (4).

The first formula of the weekday theorem expresses the definition of a weekday. The second formula saves an unnecessary multiplication by recognizing that $(365)_7 = 1$. The third formula is convenient for computations that use 16 bit registers and 2's complement arithmetic. For such computation, if y is restricted to the range $y < |28{,}000|$, then at the maximum value of y, $n = 6783$ and $y + n = 34{,}782$, which overflows, since the arithmetic is restricted to values less than 32,768, the 15th power of 2. The restriction on y does not affect the universality of the calendar

generator, since any year outside the range can be reduced modulo 28,000 prior to computation.

To avoid the idiosyncracies of leap years, we propose, as a first approximation to the actual yearday, a **standardized yearday** s that is a function of (m, d) only,

$$s = S(m) + d - 1 \tag{6}$$

where S is a function of m only. In this definition, $(y, 1, 1)$ corresponds to standardized yearday 0 and actual yearday 0. Once s has been determined, the **actual yearday**, a, is given by

$$a = s + H(q, s), \tag{7}$$

where q is the kind of year, and $H(q, s)$ is a correction term. For computational purposes, we take $q = 0$ for non-leap years, and $q \neq 0$ for leap years.

A candidate for S is

$$S(m) = \left\lfloor \frac{\gamma(m-1) + \beta}{\alpha} \right\rfloor$$

where α, β, γ are integer parameters, selected so that

$$L(m, q) = S(m+1) - S(m)$$

for all months except $m = 2$ (February), whose length clearly depends on q, and must include the correction function H. The closest fit for $m = 2$ gives $S(3) - S(2) = 30$. Using this fit, a possible choice for H is

$$H(q, s) = -2 \cdot I_{\bar{q}} \cdot I_{s > 58} - I_q \cdot I_{s > 59},$$

where I is the indicator function; I_p is 1 if p is *true* and 0 if p is *false*.

We finally have a functional form for month length,

$$L(m, q) = S(m+1) - S(m) - I_{m=2}(2I_{\bar{q}} + I_q). \tag{8}$$

The fact that $m + 1$ is invalid for $m = 12$ merely imposes a restriction on the scope of S.

Like Stone, we have not been completely successful in our attempt to find a tableless method, as the indicator functions are equivalent to a table, albeit a small one. The introduction of two extra days in February is the only algorithmic trick; hence the title of this paper.

The inverse function of S is

$$M(s) = \left\lfloor \frac{as + b}{g} \right\rfloor,$$

where a, b, g are found in the same way as α, β, γ in the case of $S(m)$. $M(s)$ transforms any day of the year into the corresponding month. The inversion is not required for calendar generation, but is useful in many date problems.

It was mentioned previously that $S(13)$ is required for month length calculation. Hence S places 13 constraints on α, β, γ:

$$S(1) = 0; \quad S(2) = 31; \quad S(3) = 61; \ \ldots \ ; \ S(13) = 367.$$

Similarly, the inversion of S places 24 constraints on a, b, g:

$$M(0) = M(30) = 1; \quad M(31) = M(60) = 2; \ \ldots \ ; \ M(336) = M(366) = 12.$$

Aided by a computer, we solve by simple enumeration. The smallest $[\alpha, \beta, \gamma]$ is $[7, 3, 214]$, a solution that is also valid for the inverse. For languages and machines that allow shifting of operands, it is efficient to do integer multiplication and division by shifting when possible; hence we recommend values $[16, 8, 489]$ for $[\alpha, \beta, \gamma]$, and $[16, 6, 489]$ for $[a, b, g]$.

We now have all the theory required to find the weekday W corresponding to any given date: add the weekday corresponding to the first day of the year to the actual yearday and reduce the result modulo 7, giving

$$W(y, m, d) = (w(y, n) + a(s, q))_7, \tag{9}$$

where w, n, a, s, q have been defined in formulas (2) to (6).

As an example of the use of these formulas, we shall determine the day of the week corresponding to the fifteenth of October 1582, the date on which the new style calendar was inaugurated. Because dates are such a vital and familiar item of modern human activity, most people can at a glance deduce that the given date is valid, and that 1582 is not a leap year. A computer however must go through formal algorithms to do so, as discussed earlier. Having decided that the date is valid, the computer then determines the day of the week corresponding to the first day of the year. Substituting in formula (3), it finds that there have been 383 leap years since year 0. Substituting in (5), 1582 started on weekday 5, a Friday. Formula (6) gives 289 for the standardized day of the year. Formula (7) gives 287 for the actual day of the year. Finally, formula (9) gives the required weekday 5, also a Friday. Readers can check the result against the historic old style calendar used by England and her colonies. They will find that October 15, 1582 corresponds to a Monday, and subtracting the ten deleted days gives Friday, conforming to our result.

The preceding theory encourages exceedingly simple computer algorithms when the algorithms are expressed in a reasonably rich higher level computer language (such as C, Pascal, Algol). An obvious application is the generation of calendars, but more important is the efficient storage and retrieval of dates in data bases, as almost all data base entries are dated. Dates can be stored as *century day numbers* (cdn), with cdn 1 corresponding to the first day of this century, and cdn 36,524 corresponding to the final day. Hence storage requirement is reduced significantly, as the cdn is well within the capacity of a 16-bit register for all dates of this century. Alternatively we could use an *absolute day number* (adn), where adn 0 corresponds to the first day of century 0, and adn 10,226,782 corresponds to the final day of the 28,000-year cycle. Age calculations become remarkably simple, as age in days is the simple difference between two adn's or two cdn's. Absolute day numbers allow a simpler form of the weekday function, but require 24-bit arithmetic.

Various countries adopted the Gregorian calendar at different times. Dates for pre-adoptive eras differ from those of the extrapolated Gregorian calendar. Even for these cases, the suggested method is useful, as it provides a convenient standard that can easily be corrected to apply to any particular era.

References

[1] R. A. Stone, Algorithm 398, collected algorithms, CACM (1970).

[2] Encyclopaedia Britannica, 1968.